Solving the Incremental Satisfiability Problem

John N. Hooker
Carnegie Mellon University, john@hooker.tepper.cmu.edu

Published In
Solving the Incremental Satisfiability Problem *

J. N. Hooker
Graduate School of Industrial Administration
Carnegie Mellon University, Pittsburgh, PA 15213 USA

March 1991
Revised March 1992

Abstract

Given a set of clauses in propositional logic that have been found satisfiable, we wish to check whether satisfiability is preserved when the clause set is incremented with a new clause. We describe an efficient implementation of the Davis-Putnam-Loveland algorithm for checking satisfiability of the original set. We then show how to modify the algorithm for efficient solution of the incremental problem, which is NP-complete. We also report computational results.

Suppose that a given proposition does not follow from a knowledge base encoded in propositional logic. This can be checked by adding the proposition’s denial to the knowledge base and verifying that the resulting set of propositions is satisfiable. We wish to determine whether the proposition follows when the knowledge base is augmented by one or more new propositions. Rather than re-solve the satisfiability problem from scratch, we propose to use information gained while solving the original problem, so as to speed the solution of the new problem.

We therefore address the incremental satisfiability problem of propositional logic: given that a set $S$ of propositional clauses is satisfiable, check whether $S \cup \{C\}$ is satisfiable for a given clause $C$. Our investigation was occasioned by efforts to solve logic circuit verification problems [6], but it has application whenever one wishes to check again for logical inferences after enlarging a propositional knowledge base. It also plays a critical role in the solution of inference problems in first-order predicate logic [4, 7].

Our approach is first to solve the original satisfiability problem with the classical Davis-Putnam-Loveland (DPL) algorithm [1, 3, 9, 11, 12]. We then solve the incremented problem with a modified DPL algorithm that takes advantage of the information in the data structure generated during the solution of the original problem. We choose DPL because, as has been reported elsewhere [2, 5, 10], it is quite competitive with more recent algorithms when it is properly implemented, and it has the advantage of simplicity. But satisfiability algorithms other than DPL can be modified in similar fashion.

The superior performance of our DPL implementation relies critically on the use of an intelligent branching rule (that of Jeroslow and Wang) and an efficient data structure for reconstructing the problem at a node after backtracking. The data structure is straightforward, but since it has not been described elsewhere, we will provide a detailed statement of the implementation of DPL we have found to be very efficient. We conclude by reporting computational tests that demonstrate the advantage of our approach.

The incremental satisfiability problem is clearly NP-complete, because one can solve a classical satisfiability problem on \( m \) clauses by solving at most \( m \) incremental problems.

1 The DPL Algorithm

If \( x_j \) denotes an atomic proposition, then \( x_j \) and its denial \( \neg x_j \) are literals. A clause is a nontautologous disjunction of distinct literals. A set of clauses is satisfiable if some assignment of truth values to atomic propositions makes all of the clauses true. An empty clause contains no literals and is necessarily false.

The DPL algorithm makes essential use of the unit resolution procedure, also known as forward chaining. Unit resolution is sound but, unlike DPL, incomplete. The procedure is applied to a given set \( S \) of clauses as follows. Look for a unit clause (a clause containing just one literal \( \ell \)) in \( S \), and fix the value of \( \ell \) to true. Remove all clauses from \( S \) containing \( \ell \) and all occurrences of \( \ell \)'s negation from the remaining clauses, and repeat. The procedure terminates when \( S \) is empty (indicating that the given clause set is satisfiable) or \( S \) contains an empty clause (indicating that the given set is unsatisfiable)—or, if neither of these occurs, when \( S \) contains no unit clause (in which case the satisfiability issue is unsettled). An empty clause
is obtained when a literal is removed from a unit clause.

The DPL procedure searches a binary tree with the following structure. Every node of the tree is associated with a set of clauses, and the root node is associated with the set $S$ to be checked for satisfiability. A nonleaf node associated with clause set $S'$ has two children associated respectively with sets $S'' \cup \{x_j\}$ and $S'' \cup \{-x_j\}$, where $x_j$ is any variable that occurs in $S''$, and $S''$ is the result of applying the unit resolution procedure to $S'$. (The unit clauses $x_j$ and $-x_j$ that are added to $S''$ are known as branch cuts.) $S$ is unsatisfiable if and only if there exists such a tree in which every leaf can be falsified; that is, every leaf is associated with a clause set for which unit resolution yields the empty clause. (Optionally, the unit resolution procedure can be augmented so as to fix monotone variables before performing any resolutions. That is, a variable $x_j$ is set to true if it is posited in every occurrence in $S'$ and is set to false if it is negated in every occurrence.)

DPL normally traverses the tree depth-first, since this requires that fewer nodes be kept in storage at any one time (namely, those along a path from the current node to the root). Traversal begins at the root node. At any node associated with clause set $S'$ there are three cases. a) The unit resolution procedure applied to $S'$ generates an empty set of clauses. Then the algorithm terminates, because $S$ is satisfiable. b) Unit resolution generates the empty clause. The algorithm backtracks along the path to the root until it reaches a node with only one child, whereupon it generates the other child and continues at the new node. If it reaches the root without finding such a node, it terminates, because $S$ is unsatisfiable. c) Neither of these occurs, in which case the algorithm branches. It generates a child associated with either $S'' \cup \{x_j\}$ or $S'' \cup \{-x_j\}$, where $x_j$ occurs in $S''$. It then continues at the new node.

The Jeroslow-Wang branching rule [8] indicates which child should be generated first when the algorithm branches. It says roughly that the literal added to $S''$ should occur in a large number of short clauses in $S''$. Let the length of a clause be the number of literals in it. If $v$ represents a truth value (0 or 1), define the function

$$w(S'', j, v) = \sum_{k=1}^{\infty} N_{j,k,v} 2^{-k},$$

where $N_{j,k,v}$ is the number of clauses of length $k$ that contain $x_j$ (if $v = 1$) or $-x_j$ (if $v = 0$). If $(j^*, v^*)$ maximizes $w(S'', j, v)$, then we take the $S'' \cup \{x_j^*\}$
branch first if \( v^* = 1 \), and otherwise take the \( S'' \cup \{-x_j,\} \) branch first. The rationale is that \( w(S'', j, v) \) estimates the probability that a random truth assignment will falsify one of the clauses eliminated when one branches on \( x_j \). Thus by maximizing \( w(S'', j, v) \) we also maximize the probability that a random truth assignment will satisfy all of the remaining clauses. This can lead to a satisfying solution earlier in the search process, if \( S \) is satisfiable. Conversely, if \( S \) is unsatisfiable, then fixing the value of literals in short clauses tend to generate more unit clauses and therefore to allow unit resolution to detect inconsistency more quickly.

When backtracking to a node, we must restore the clause set \( S' \) corresponding to that node. One option is to store the problem associated with every unfathomed node, but this may require excessive storage. Another option is to store only the original problem set \( S \), add to \( S \) all the branch cuts between the current node and the root, and perform unit resolution on the resulting set to obtain \( S' \). But this is quite time consuming. A better alternative is to maintain for each clause \( i \) the highest level \( a_i \), in the path from the root to the current node, at which the clause is still in \( S' \) (the root is level 1). We also maintain for each variable \( x_j \) the highest level \( b_i \) at which the \( x_j \) still occurs in \( S' \). A variable that is no longer in \( S' \) at level \( k \) may have been removed for either of two reasons: its value was fixed by unit resolution, or all the clauses containing it were removed by unit resolution (or both).

In the accompanying statement of the DPL procedure (Algorithm 1), we let \( L_k \) be the branch cut added to obtain the left child of the node last visited at level \( k \) and \( R_k \) the cut added to obtain the right child, with \( R_k = 0 \) if no right child has been generated. \( T \) contains the variables fixed to true at the current node, and \( F \) those fixed to false. If a solution is found, it is recovered by setting the variables currently in \( T \) to true and those in \( F \) to false; each of the remaining variables may be set to either true or false. We use a procedure UPDATE to keep track of the clause set associated with the current node, and a procedure RESTORE to restore the problem associated with the current node.

The algorithm is written in pseudo-Pascal, so that no procedure (except DPL) is executed unless called by a subsequent procedure. Execution therefore starts at the Begin statement following the end of the BUILD TREE procedure.

Our FORTRAN implementation of the algorithm stores each clause in the form of two linked lists. One contains the indices of positive literals in ascending order, and the other similarly lists the negative literals. Pointers
to these lists are stored in a linked list, in ascending order of clause indices. Variables and clauses can therefore be quickly added or deleted, and the running time for RESTORE and UPDATE is linear in the number of literals. The values of the Jeroslow-Wang function $w(S', j, v)$ are computed simultaneously for all $(j, v)$ on a single pass through the data structure. The sets $T$ and $F$ are maintained simply as an array that records true, false or undetermined for each variable.

2 An Incremental DPL Algorithm

Consider the search tree that the above DPL algorithm generates when testing $S$ for satisfiability. When we add a new clause $C$ to $S$, the clause set $S'$ associated with any fathomed node of this tree remains unsatisfiable. So, there is no point in looking again at any part of the tree already generated, except the path from the root to the last node examined. We simply continue to build the tree, beginning at an appropriate point along this path, and keeping in mind that $C$ must also be satisfied. We must also update the data structure to show at what unfathomed nodes $C$ remains in the problem.

More precisely, when we initially determine that $S$ is satisfiable, we save the data structure by saving the numbers $a_i$, $b_j$, the sets $T$, $F$, the cuts $L_k$, $R_k$, and the level $k$ in the search tree at which the search terminates. Then to test $S \cup \{C\}$ for satisfiability, we first check whether the variables fixed at level $k$ satisfy or falsify $C$. If they satisfy $C$, we note that $S \cup \{C\}$ is satisfiable and quit. If they falsify $C$, we backtrack, along the path from the current node to the root, to the node at which $C$ is first falsified, and resume the DPL algorithm at that point. If the fixed variables leave the truth value of $C$ undetermined, we branch on one of the variables in $C$ (unless only one variable in $C$ is unfixed, in which case we simply fix its value.) We then note that $S \cup \{C\}$ is satisfiable and quit. In all of these cases, we update the data structure to indicate at which node $C$ is eliminated from the problem.

In Algorithm 2, which is a statement incremental DPL, $k_0$ is the level, along the path from the root to the current node, at which $C$ is first falsified (if at all), and $k_1$ the level at which it is first satisfied (if at all). The algorithm can be applied repeatedly as new clauses are added.
3 Computational Results

Algorithms 1 and 2 were tested on part of a collection of satisfiability problems assembled by F. J. Radermacher [10, 13] for establishing benchmarks. We solved the first two problems from each of 14 groups of problems, and the results appear in Table 1. Problem groups ulmxxxr0, ulmxxxr1 and ulmxxxr2 encode systems of linear equations modulo 2. Groups real1x12 and real2x12 are stuck-at-zero problems in VLSI design. Groups jnhx, jnh2xx and jnh3xx are random problems intentionally generated with parameters set to yield hard problems. Groups ulmbcxxx, ulmbpxxx and ulmbxxxx are chessboard problems. Group twoinvrx consists of logic circuit construction problems, group holex of pigeon hole problems, and group nodxcolx of graph coloring problems. Table 1 shows the number $n$ of atomic propositions in each problem. A more detailed description of the problems can be found in [2, 10, 13].

For each problem, the order of clauses was randomized, and a subproblem consisting of clauses $1, \ldots, k$ of the original problem was solved for each $k = 1, \ldots, m$. When the original problem is satisfiable, $m$ is the number of clauses in the problem, and otherwise $m$ is the number of clauses that had been added when the subproblem first became unsatisfiable. The series of subproblems was solved once by solving each subproblem from scratch using the DPL algorithm stated above (the results are marked DPL in Table 1). The subproblems were then re-solved by repeated application of the incremental DPL algorithm (results marked IDPL). Table 1 shows results not only for the entire series of subproblems $1, \ldots, m$ (left side of the table), but for the last problem alone as well (right side). It also indicates by 'Y' or 'N' whether the last subproblem is satisfiable.

The DPL algorithm is the same as tested in [2, 10]. It and the incremental algorithm were coded in FORTRAN and run on a Sun Sparc Station 330 operating under SunOS 4.1.

The incremental algorithm is substantially faster for 22 out of 28 problems. In at least half the problems, incremental DPL solved all the subproblems in about the same time it took DPL to solve only the last subproblem.

When choosing a variable on which to branch, DPL can apply the Jeroslow-Branching rule to the entire problem, whereas incremental DPL can look only at the clauses that have been added so far. One might therefore expect incremental DPL to use a less efficient branching order than DPL, and this could offset its advantage of using the search tree accumulated in prior solutions. But as we noted, this happened for only 6 of the 28
problems. In fact one would not expect incremental DPL to perform well on at least two of the six, namely the pigeon hole problems, since for these the subproblems are quite easy until the very last clause is added. Incremental DPL can also result in more efficient branching order than DPL, since for three of the test problems, incremental DPL takes less time to solve all the incremental problems than DPL takes to solve the last one only.

References


Algorithm 1 (DPL)

Procedure DPL; \{Check clause set $S$ for satisfiability.\}

Procedure UNIT RESOLUTION; \{$S'$ will contain results of unit resolution.\}

Begin
  If $S' = \emptyset$ or $S'$ contains the empty clause, stop;
  While $S'$ contains a unit clause $C$ do
    Begin
      If $S' = \emptyset$ or $S'$ contains the empty clause, stop;
      Else
        If the variable $x_j$ in $C$ is positive, then
          Begin
            Add $x_j$ to $T$;
            Remove from $S'$ all clauses containing literal $x_j$ and all occurrences of literal $\neg x_j$;
          End
        Else
          Begin
            Add $x_j$ to $F$;
            Remove from $S'$ all clauses containing literal $\neg x_j$ and all occurrences of literal $x_j$;
          End;
    End;
End UNIT RESOLUTION;

Procedure UPDATE; \{Record variables and clauses that are in $S'$.\}

Begin
  For each clause $i$ in $S'$ do
    Begin
      Let $a_i := k$;
      For each variable $x_j$ in clause $i$ do
        Let $b_j := k$;
    End;
End UPDATE;

\{Continued on next page.\}
Procedure RESTORE; \{Reconstruct problem at current level $k$ node.\}

Begin

For all $j$ such that $b_j \geq k$ do

Begin

Remove $x_j$ from $T$ or $F$ (whichever contains it);
Let $b_j := k$;

End;

Let $S' := \emptyset$;

For all clauses $i$ in $S$ do

If $a_i \geq k$ then

Begin

Let $a_i := k$;

Delete from clause $i$ all literals involving a variable $x_j$ for which $b_j < k$, and add the resulting clause to $S'$;

End;

End RESTORE;

\{Continued on next page.\}
Procedure BUILD TREE; \{Conduct tree search beginning at level \(k\).\}

Begin

While \(k > 0\) do

Begin

Perform UNIT RESOLUTION;
Perform UPDATE; \{Record variables and clauses left in the problem.\}
If \(S'\) is empty, then
Stop, because \(S\) is satisfiable;
Else
If \(S'\) contains the empty clause, then
\{Backtrack to a node with no right child.\}
Begin
Let \(R_k := 1\);
While \(k > 0\) and \(R_k \neq 0\) do
\(k := k - 1;\)
If \(k > 0\) then
Begin
\{Reconstruct the problem at this node and then generate right child.\}
Perform RESTORE;
Let \(R_k := \neg R_k\) and \(S' := S' \cup \{R_k\}\);
Let \(k := k + 1;\)
End;
End
Else \{Branch as recommended by Jeroslow-Wang heuristic.\}
Begin
Pick \(j^*, v^*\) so that
\(w(S', j^*, v^*) = \max_{j, v} \{w(S', j, v)\};\)
If \(v^* = 1\) then let \(L_k := x_{j^*}\);
Else let \(L_k := \neg x_{j^*};\)
Let \(R_k := 0, S' := S' \cup \{L_k\}\) and \(k := k + 1;\)
End;
Stop, because \(S\) is unsatisfiable;
End BUILD TREE;

Begin \{Execution starts here.\}

Begin with a set \(S\) of clauses containing variables \(x_1, \ldots, x_n;\)
Let \(S' := S, k := 0, T := \emptyset, F := \emptyset, \) and perform UPDATE;
Set the level \(k := 1;\)
Perform BUILD TREE;
End DPL.
Algorithm 2 (Incremental DPL)

Procedure INCREMENTAL DPL; \{Check $S \cup \{C\}$ for satisfiability.\}

Procedure SCAN NEW CLAUSE;
\{Check which literals in $C$ are fixed, and let $C'$ contain those remaining.
Determine at what level in the tree $C$ is first satisfied or falsified (if either)
by fixed literals.\}

Begin
For each literal $\ell$ in $C'$ do
Begin
Let $x_j$ be the variable in $\ell$;
\{If $x_j$ is true and occurs positively in $C$, then $C$ is already satisfied
at a level no higher than where $x_j$ is fixed.
But if $x_j$ is negated in $C$, then it is removed from $C$, and
$C$ is already falsified at a level no lower than where $x_j$ is fixed.\}
If $x_j \in T$ then
Begin
If $\ell = x_j$ then
Let $k_1 := \min\{k_1, b_j\}$
Else
Remove $x_j$ from $C'$ and let $k_0 := \max\{k_0, b_j\}$;
End
\{Analogously if $x_j$ is false.\}
If $x_j \in F$ then
Begin
If $\ell = \neg x_j$ then
Let $k_1 := \min\{k_1, b_j\}$
Else
Remove $\neg x_j$ from $C'$ and let $k_0 := \max\{k_0, b_j\}$;
End;
End;
End SCAN NEW CLAUSE;
\{Continued on next page.\}
Procedure BRANCH; \{Branch on an undetermined variable in \( C \).\}

Begin

Let \( a_i := k \);

If \( C' \) is a unit clause, then

\{In this case, there is no need to branch, because the remaining literal must be true.\}

Begin

Let \( x_j \) be the variable in \( C' \);
If \( C' = x_j \) then add \( x_j \) to \( T \)
Else add \( x_j \) to \( F \);

End

Else

\{Otherwise create a new child node, in preparation for the next time INCREMENTAL DPL is executed.\}

Begin

For each variable \( x_j \) in \( C' \) do

If \( x_j \not\in T \cup F \) then \( b_j := k \);
Pick a literal \( \ell \) in \( C' \) and let \( x_j \) be the variable in \( \ell \);
If \( \ell = x_j \) then

Add \( x_j \) to \( T \) and set \( L_k := x_j \), \( R_k := 0 \);
Else

Add \( x_j \) to \( F \) and set \( L_k := \neg x_j \), \( R_k := 0 \);
Let \( a_i := k \), \( k := k + 1 \);

End;

Stop, because \( S \) is satisfiable;

End BRANCH;

\{Continued on next page.\}
Begin \( \{ \text{Execution begins here.} \} \)
Let \( k_0 := 0, k_1 := \infty, C' := C \); Let \( C \) be numbered clause \( i \);
Let \( S := S \cup \{ C \} \);
\( \{ \text{Check which literals of } C \text{ have been fixed, let } C' \text{ contain those remaining,} \)
\( \text{and determine at what level } C \text{ is first satisfied or falsified (if either).} \}
Perform SCAN NEW CLAUSE;
\( \{ \text{If } C \text{ is already satisfied, record removal of } C \text{ at the level at which} \)
\( \text{it is first satisfied, and stop.} \}
If \( k_1 < \infty \) then
Begin
For each variable \( x_j \) in \( C \) do
If \( x_j \not\in T \cup F \) then let \( b_j := \max\{ b_j, k_1 \} \);
Set \( a_i := k_1 \);
Stop, because \( S \cup \{ C \} \) is satisfiable;
End
\( \{ \text{If } C \text{ is already falsified, backtrack to the level at which it is first falsified.} \}
Else
If \( C' \) is empty then
Begin
For each variable \( x_j \) in \( C \) do
If \( x_j \not\in T \cup F \) then let \( b_j := \max\{ b_j, k_0 \} \);
Let \( k := k_0 + 1, R_k := 1, a_i := k_0 \);
While \( k > 0 \) and \( R_k \neq 0 \) do
\( k := k - 1 \);
\{ \text{Restore problem at node backtracked to, generate the} \)
\text{right child, and continue the tree search in DPL.} \}
If \( k > 0 \) then
Begin
Perform RESTORE;
Let \( R_k := \neg L_k \) and \( S' := S' \cup \{ R_k \} \);
Let \( k := k + 1 \);
Perform BUILD TREE;
End
\{ \text{S is unsatisfiable if backtracking is all the way to root.} \}
Else
Stop, because \( S \) is unsatisfiable;
End
Else
\{ \text{Branch if } C \text{ is neither satisfied or falsified by fixed literals.} \}
Perform BRANCH;
End INCREMENTAL DPL.
Table 1: Computational Results.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Size</th>
<th>All Increments</th>
<th>Last Increment Only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n m</td>
<td>No. Nodes DPL IDPL</td>
<td>CPU Sec. DPL IDPL</td>
</tr>
<tr>
<td>ulm027r0</td>
<td>25 64</td>
<td>638 83 1.16 0.06</td>
<td>Y 7 1 0.04 0.00</td>
</tr>
<tr>
<td>ulm027r1</td>
<td>23 64</td>
<td>671 78 1.31 0.07</td>
<td>Y 21 1 0.08 0.00</td>
</tr>
<tr>
<td>ulm027r2</td>
<td>25 64</td>
<td>669 86 1.16 0.08</td>
<td>Y 7 1 0.04 0.00</td>
</tr>
<tr>
<td>ulm054r0</td>
<td>50128</td>
<td>2502 299 10.9 0.34</td>
<td>Y 13 1 0.16 0.00</td>
</tr>
<tr>
<td>ulm054r1</td>
<td>46128</td>
<td>3143 291 13.4 0.42</td>
<td>Y 265 1 1.33 0.00</td>
</tr>
<tr>
<td>ulm054r2</td>
<td>50128</td>
<td>2492 336 10.7 0.57</td>
<td>Y 13 1 0.17 0.00</td>
</tr>
<tr>
<td>real1a12</td>
<td>18275</td>
<td>1221 273 7.40 0.05</td>
<td>Y 3 1 0.05 0.00</td>
</tr>
<tr>
<td>real1b12</td>
<td>15110</td>
<td>471 124 1.03 0.05</td>
<td>Y 3 1 0.02 0.00</td>
</tr>
<tr>
<td>real2a12</td>
<td>18275</td>
<td>1226 282 7.32 0.07</td>
<td>Y 3 1 0.06 0.00</td>
</tr>
<tr>
<td>real2b12</td>
<td>15110</td>
<td>471 124 1.01 0.08</td>
<td>Y 3 1 0.02 0.00</td>
</tr>
<tr>
<td>jnh201</td>
<td>100800</td>
<td>35087 6989 2419 266</td>
<td>Y 32 1 8.06 0.00</td>
</tr>
<tr>
<td>jnh202</td>
<td>100778</td>
<td>38623 101897 3095 12036</td>
<td>N 93 8325 50.1 1178.</td>
</tr>
<tr>
<td>jnh1</td>
<td>100850</td>
<td>41083 16347 2684 998</td>
<td>Y 126 1 18.7 0.00</td>
</tr>
<tr>
<td>jnh2</td>
<td>100838</td>
<td>35099 9929 2971 11.21</td>
<td>N 221 411 81.4 85.0</td>
</tr>
<tr>
<td>jnh301</td>
<td>100900</td>
<td>44619 198730 3248 27770</td>
<td>Y 360 1 72.4 0.00</td>
</tr>
<tr>
<td>jnh302</td>
<td>100756</td>
<td>35621 3920 2286 683.</td>
<td>N 109 1665 65.4 634.</td>
</tr>
<tr>
<td>ulmbc024</td>
<td>24164</td>
<td>2122 185 5.69 0.09</td>
<td>Y 10 1 0.09 0.00</td>
</tr>
<tr>
<td>ulmbc040</td>
<td>40340</td>
<td>6713 364 47.9 0.31</td>
<td>Y 24 1 0.42 0.00</td>
</tr>
<tr>
<td>ulimbp048</td>
<td>4899</td>
<td>2455 2820 9.33 16.1</td>
<td>N 505 406 3.5 2.79</td>
</tr>
<tr>
<td>ulimbp070</td>
<td>70161</td>
<td>9437 29212 69.7 252.</td>
<td>N 4671 28933 42.2 251.</td>
</tr>
<tr>
<td>ulimbs040</td>
<td>4070</td>
<td>1197 139 2.39 0.11</td>
<td>Y 11 1 0.06 0.00</td>
</tr>
<tr>
<td>ulimbs060</td>
<td>60120</td>
<td>2997 189 12.2 0.22</td>
<td>Y 16 1 0.18 0.00</td>
</tr>
<tr>
<td>twoinvt</td>
<td>30 79</td>
<td>921 141 2.32 0.14</td>
<td>Y 5 1 0.06 0.00</td>
</tr>
<tr>
<td>twoinvr1</td>
<td>30 81</td>
<td>975 116 2.58 0.11</td>
<td>Y 15 1 0.21 0.00</td>
</tr>
<tr>
<td>hole6</td>
<td>42133</td>
<td>9665 14441 32.8 82.1</td>
<td>N 6491 14278 22.4 82.5</td>
</tr>
<tr>
<td>hole7</td>
<td>56204</td>
<td>71826 184714 370 1550.</td>
<td>N 65561184482 326. 1550.</td>
</tr>
<tr>
<td>nod5col4</td>
<td>40128</td>
<td>2956 394 12.4 2.21</td>
<td>N 205 169 2.09 1.85</td>
</tr>
<tr>
<td>nod6col4</td>
<td>60226</td>
<td>7554 709 73.2 9.60</td>
<td>N 263 99 7.85 2.34</td>
</tr>
</tbody>
</table>

*The computer does not measure CPU times less than 0.01 second.*